Privacy Pass Issuance Protocol

Abstract

This document specifies two variants of the two-message issuance protocol for Privacy Pass tokens: one that produces tokens that are privately verifiable using the issuance private key, and another that produces tokens that are publicly verifiable using the issuance public key.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 11 June 2023.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (https://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in...
1. Introduction

The Privacy Pass protocol provides a privacy-preserving authorization mechanism. In essence, the protocol allows clients to provide cryptographic tokens that prove nothing other than that they have been created by a given server in the past [ARCHITECTURE].

This document describes the issuance protocol for Privacy Pass built on [HTTP]. It specifies two variants: one that is privately verifiable using the issuance private key based on the oblivious pseudorandom function from [OPRF], and one that is publicly
This document does not cover the Privacy Pass architecture, including choices that are necessary for ensuring that client privacy leaks. This information is covered in [ARCHITECTURE].

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The following terms are used throughout this document.

*Client: An entity that provides authorization tokens to services across the Internet, in return for authorization.

*Issuer: A service that produces Privacy Pass tokens to clients.

*Private Key: The secret key used by the Issuer for issuing tokens.

*Public Key: The public key used by the Issuer for issuing and verifying tokens.

Unless otherwise specified, this document encodes protocol messages in TLS notation from [TLS13], Section 3. Moreover, all constants are in network byte order.

3. Configuration

Issuers MUST provide two parameters for configuration:

1. Issuer Request URI: A token request URL for generating access tokens. For example, an Issuer URL might be https://issuer.example.net/example-token-request. This parameter uses resource media type "text/plain".

2. Issuer Public Key values: An Issuer Public Key for an issuance protocol.

The Issuer parameters can be obtained from an Issuer via a directory object, which is a JSON object [RFC8259], Section 4 whose values are other JSON values [RFC8259], Section 3 for the parameters.
<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>issuer-request-uri</td>
<td>Issuer Request URI resource percent-encoded URL string, represented as a JSON string [RFC8259], Section 7</td>
</tr>
<tr>
<td>token-keys</td>
<td>List of Issuer Public Key values, each represented as JSON objects [RFC8259], Section 4</td>
</tr>
</tbody>
</table>

Each "token-keys" JSON object contains the following fields and corresponding raw values.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>token-type</td>
<td>Integer value of the Token Type, as defined in Section 8.2, represented as a JSON number [RFC8259], Section 6</td>
</tr>
<tr>
<td>token-key</td>
<td>The base64url encoding of the public key for use with the issuance protocol, including padding, represented as a JSON string [RFC8259], Section 7</td>
</tr>
</tbody>
</table>

Issuers MAY advertise multiple token-keys for the same token-type to support key rotation. In this case, Issuers indicate preference for which token key to use based on the order of keys in the list, with preference given to keys earlier in the list.

Altogether, the Issuer's directory could look like:

```json
{
  "issuer-request-uri": "https://issuer.example.net/example-token-request",
  "token-keys": [
    {
      "token-type": 2,
      "token-key": "MI...AB",
    },
    {
      "token-type": 2,
      "token-key": "MI...AQ",
    }
  ]
}
```

Issuer directory resources have the media type "application/json" and are located at the well-known location /.well-known/token-issuer-directory; see Section 8.1 for the registration information for this well-known URI.
4. Token Challenge Requirements

Clients receive challenges for tokens, as described in [AUTHSCHEME]. The basic token issuance protocols described in this document can be interactive or non-interactive, and per-origin or cross-origin.

5. Issuance Protocol for Privately Verifiable Tokens

The Privacy Pass issuance protocol is a two message protocol that takes as input a TokenChallenge from the redemption protocol [AUTHSCHEME], Section 2.1 and produces a Token [AUTHSCHEME], Section 2.2, as shown in the figure below.

Issuers provide a Private and Public Key, denoted skI and pkI, respectively, used to produce tokens as input to the protocol. See Section 5.5 for how this key pair is generated.

Clients provide the following as input to the issuance protocol:

*Issuer name, identifying the Issuer. This is typically a host name that can be used to construct HTTP requests to the Issuer.

*Issuer Public Key pkI, with a key identifier token_key_id computed as described in Section 5.5.

*Challenge value challenge, an opaque byte string. For example, this might be provided by the redemption protocol in [AUTHSCHEME].

Given this configuration and these inputs, the two messages exchanged in this protocol are described below. This section uses notation described in [OPRF], Section 4, including SerializeElement and DeserializeElement, SerializeScalar and DeserializeScalar, and DeriveKeyPair.

5.1. Client-to-Issuer Request

The Client first creates a context as follows:

\[
\text{client\_context} = \text{SetupVOPRF\_Client(0x0004, pkI)}
\]
Here, 0x0004 is the two-octet identifier corresponding to the OPRF(P-384, SHA-384) ciphersuite in [OPRF]. SetupVOPRFClient is defined in [OPRF], Section 3.2.

The Client then creates an issuance request message for a random value nonce with the input challenge and Issuer key identifier as described below:

nonce = random(32)
challenge_digest = SHA256(challenge)
token_input = concat(0x0001, nonce, challenge_digest, token_key_id)
blind, blinded_element = client_context.Blind(token_input)

The Blind function is defined in [OPRF], Section 3.3.2. If the Blind function fails, the Client aborts the protocol. The Client stores the nonce and challenge_digest values locally for use when finalizing the issuance protocol to produce a token (as described in Section 5.3).

The Client then creates a TokenRequest structured as follows:

```c
struct {
  uint16_t token_type = 0x0001; /* Token type VOPRF(P-384, SHA-384) */
  uint8_t truncated_token_key_id;
  uint8_t blinded_msg[Ne];
} TokenRequest;
```

The structure fields are defined as follows:

**"token_type"** is a 2-octet integer, which matches the type in the challenge.

**"truncated_token_key_id"** is the least significant byte of the token_key_id in network byte order (in other words, the last 8 bits of token_key_id).

**"blinded_msg"** is the Ne-octet blinded message defined above, computed as SerializeElement(blinded_element). Ne is as defined in [OPRF], Section 4.

The values token_input and blinded_element are stored locally and used later as described in Section 5.3. The Client then generates an HTTP POST request to send to the Issuer, with the TokenRequest as the content. The media type for this request is "application/private-token-request". An example request is shown below.
Upon receipt of the request, the Issuer validates the following conditions:

* The TokenRequest contains a supported token_type.

* The TokenRequest.truncated_token_key_id corresponds to the truncated key ID of a Public Key owned by the issuer.

* The TokenRequest.blinded_msg is of the correct size.

If any of these conditions is not met, the Issuer MUST return an HTTP 400 error to the client.

5.2. Issuer-to-Client Response

Upon receipt of a TokenRequest, the Issuer tries to deserialize TokenRequest.blinded_msg using DeserializeElement from Section 2.1 of [OPRF], yielding blinded_element. If this fails, the Issuer MUST return an HTTP 400 error to the client. Otherwise, if the Issuer is willing to produce a token to the Client, the Issuer completes the issuance flow by computing a blinded response as follows:

server_context = SetupVOPRFServer(0x0004, skI, pkI)
evaluate_element, proof = server_context.Evaluate(skI, blinded_element)

SetupVOPRFServer is in [OPRF], Section 3.2 and Evaluate is defined in [OPRF], Section 3.3.2. The Issuer then creates a TokenResponse structured as follows:

struct {
    uint8_t evaluate_msg[Ne];
    uint8_t evaluate_proof[Ns+Ns];
} TokenResponse;

The structure fields are defined as follows:

*"evaluate_msg" is the Ne-octet evaluated messaged, computed as SerializeElement(evaluate_element).
"evaluate_proof" is the (Ns+Ns)-octet serialized proof, which is a pair of Scalar values, computed as concat(SerializeScalar(proof[0]), SerializeScalar(proof[1])), where Ns is as defined in [OPRF], Section 4.

The Issuer generates an HTTP response with status code 200 whose content consists of TokenResponse, with the content type set as "application/private-token-response".

:status = 200
content-type = application/private-token-response
content-length = <Length of TokenResponse>
<Bytes containing the TokenResponse>

5.3. Finalization

Upon receipt, the Client handles the response and, if successful, deserializes the content values TokenResponse.evaluate_msg and TokenResponse.evaluate_proof, yielding evaluated_element and proof. If deserialization of either value fails, the Client aborts the protocol. Otherwise, the Client processes the response as follows:

authenticator = client_context.Finalize(token_input, blind, evaluated_element, blinded_element, proof)

The Finalize function is defined in [OPRF], Section 3.3.2. If this succeeds, the Client then constructs a Token as follows:

struct {
  uint16_t token_type = 0x0001; /* Token type VOPRF(P-384, SHA-384) */
  uint8_t nonce[32];
  uint8_t challenge_digest[32];
  uint8_t token_key_id[32];
  uint8_t authenticator[Nk];
} Token;

The Token.nonce value is that which was sampled in Section 5.1. If the Finalize function fails, the Client aborts the protocol.

5.4. Token Verification

To verify a token, a verifier creates a VOPRF context using the Issuer Private Key, evaluates the token contents, and compares the result against the token authenticator value, as follows:

```
5.5. Issuer Configuration

Issuers are configured with Private and Public Key pairs, each denoted skI and pkI, respectively, used to produce tokens. A RECOMMENDED method for generating key pairs is as follows:

seed = random(Ns)
(skI, pkI) = DeriveKeyPair(seed, "PrivacyPass")

The key identifier for a public key pkI, denoted token_key_id, is computed as follows:

token_key_id = SHA256(concat(0x0001, SerializeElement(pkI)))

Since Clients truncate token_key_id in each TokenRequest, Issuers should ensure that the truncated form of new key IDs do not collide with other truncated key IDs in rotation.

6. Issuance Protocol for Publicly Verifiable Tokens

This section describes a variant of the issuance protocol in Section 5 for producing publicly verifiable tokens. It differs from the previous variant in that the output tokens are publicly verifiable by anyone with the Issuer public key.

This means any Origin can select a given Issuer to produce tokens, as long as the Origin has the Issuer public key, without explicit coordination or permission from the Issuer. This is because the Issuer does not learn the Origin that requested the token during the issuance protocol.

Beyond this difference, the publicly verifiable issuance protocol variant is nearly identical to the privately verifiable issuance protocol variant. In particular, Issuers provide a Private and Public Key, denoted skI and pkI, respectively, used to produce tokens as input to the protocol. See Section 6.5 for how this key pair is generated.
Clients provide the following as input to the issuance protocol:

*Issuer name, identifying the Issuer. This is typically a host name that can be used to construct HTTP requests to the Issuer.

*Issuer Public Key pkI, with a key identifier token_key_id computed as described in Section 6.5.

*Challenge value challenge, an opaque byte string. For example, this might be provided by the redemption protocol in [AUTHSCHEME].

Given this configuration and these inputs, the two messages exchanged in this protocol are described below.

6.1. Client-to-Issuer Request

The Client first creates an issuance request message for a random value nonce using the input challenge and Issuer key identifier as follows:

nonce = random(32)
challenge_digest = SHA256(challenge)
token_input = concat(0x0002, nonce, challenge_digest, token_key_id)
blinded_msg, blind_inv = rsabssa_blind(pkI, token_input)

The rsabssa_blind function is defined in [BLINDRSA], Section 5.1.1. The Client stores the nonce and challenge_digest values locally for use when finalizing the issuance protocol to produce a token (as described in Section 6.3).

The Client then creates a TokenRequest structured as follows:

```c
struct {
    uint16_t token_type = 0x0002; /* Token type Blind RSA (2048-bit) */
    uint8_t truncated_token_key_id;
    uint8_t blinded_msg[Nk];
} TokenRequest;
```

The structure fields are defined as follows:

""token_type" is a 2-octet integer, which matches the type in the challenge.

"truncated_token_key_id" is the least significant byte of the token_key_id in network byte order (in other words, the last 8 bits of token_key_id).

"blinded_msg" is the Nk-octet request defined above.
The Client then generates an HTTP POST request to send to the Issuer, with the TokenRequest as the content. The media type for this request is "application/private-token-request". An example request is shown below, where \( N_k = 512 \).

```plaintext
:method = POST
:scheme = https
:authority = issuer.example.net
:path = /example-token-request
:accept = application/private-token-response
:cache-control = no-cache, no-store
:content-type = application/private-token-request
:content-length = <Length of TokenRequest>

<Bytes containing the TokenRequest>
```

Upon receipt of the request, the Issuer validates the following conditions:

* The TokenRequest contains a supported token_type.

* The TokenRequest.truncated_token_key_id corresponds to the truncated key ID of a Public Key owned by the issuer.

* The TokenRequest.blinded_msg is of the correct size.

If any of these conditions is not met, the Issuer MUST return an HTTP 400 error to the Client, which will forward the error to the client.

6.2. Issuer-to-Client Response

If the Issuer is willing to produce a token token to the Client, the Issuer completes the issuance flow by computing a blinded response as follows:

\[
\text{blind\_sig} = \text{rsabssa\_blind\_sign}(skI, \text{TokenRequest.blinded\_msg})
\]

This is encoded and transmitted to the client in the following TokenResponse structure:

```c
struct {
    uint8_t blind_sig[Nk];
} TokenResponse;
```

The rsabssa_blind_sign function is defined in [BLINDRSA], Section 5.1.2. The Issuer generates an HTTP response with status code 200 whose content consists of TokenResponse, with the content type set as "application/private-token-response".
6.3. Finalization

Upon receipt, the Client handles the response and, if successful, processes the content as follows:

authenticator = rsabssa_finalize(pkI, nonce, blind_sig, blind_inv)

The rsabssa_finalize function is defined in [BLINDRSA], Section 5.1.3. If this succeeds, the Client then constructs a Token as described in [AUTHSCHEME] as follows:

```c
struct {
    uint16_t token_type = 0x0002; /* Token type Blind RSA (2048-bit) */
    uint8_t nonce[32];
    uint8_t challenge_digest[32];
    uint8_t token_key_id[32];
    uint8_t authenticator[Nk];
} Token;
```

The Token.nonce value is that which was sampled in Section 5.1. If the rsabssa_finalize function fails, the Client aborts the protocol.

6.4. Token Verification

To verify a token, a verifier checks that Token.authenticate is a valid signature over the remainder of the token input using the Issuer Public Key. The function RSASSA-PSS-VERIFY is defined in Section 8.1.2 of [RFC8017], using SHA-384 as the Hash function, MGF1 with SHA-384 as the PSS mask generation function (MGF), and a 48-byte salt length (sLen).

```c
token_authenticator_input =
    concat(Token.token_type,
        Token.nonce,
        Token.challenge_digest,
        Token.token_key_id)
valid = RSASSA-PSS-VERIFY(pkI, token_authenticator_input, Token.authenticato
```

6.5. Issuer Configuration

Issuers are configured with Private and Public Key pairs, each denoted skI and pkI, respectively, used to produce tokens. Each key pair SHALL be generated as as specified in FIPS 186-4 [DSS].
The key identifier for a keypair (skI, pkI), denoted token_key_id, is computed as SHA256(encoded_key), where encoded_key is a DER-encoded SubjectPublicKeyInfo (SPKI) object carrying pkI. The SPKI object MUST use the RSASSA-PSS OID [RFC5756], which specifies the hash algorithm and salt size. The salt size MUST match the output size of the hash function associated with the public key and token type.

Since Clients truncate token_key_id in each TokenRequest, Issuers should ensure that the truncated form of new key IDs do not collide with other truncated key IDs in rotation.

7. Security considerations

This document outlines how to instantiate the Issuance protocol based on the VOPRF defined in [OPRF] and blind RSA protocol defined in [BLINDRSA]. All security considerations described in the VOPRF and blind RSA documents also apply in the Privacy Pass use-case. Considerations related to broader privacy and security concerns in a multi-Client and multi-Issuer setting are deferred to the Architecture document [ARCHITECTURE].

Beyond these considerations, it is worth highlighting the fact that Client TokenRequest messages contain truncated token key IDs. This is done to minimize the chance that an Issuer can use distinct keys for targeting specific users. Since the key ID is truncated to a single byte, an Issuer can partition the set of Clients into at most 256 different anonymity sets. On top of this key ID space, Clients SHOULD apply some form of key consistency mechanism to help ensure they are not being given unique keys; see [CONSISTENCY] for more details.

8. IANA considerations

8.1. Well-Known 'token-issuer-directory' URI

This document updates the "Well-Known URIs" Registry [WellKnownURIs] with the following values.

<table>
<thead>
<tr>
<th>URI Suffix</th>
<th>Change Controller</th>
<th>Reference</th>
<th>Status</th>
<th>Related information</th>
</tr>
</thead>
<tbody>
<tr>
<td>token-issuer-directory</td>
<td>IETF</td>
<td>[this document]</td>
<td>permanent</td>
<td>None</td>
</tr>
</tbody>
</table>

Table 3: 'xxx' Well-Known URI

8.2. Token Type

This document updates the "Token Type" Registry from [AUTHSCHEME], Section 5.2 with the following values.
### Table 4: Token Types

<table>
<thead>
<tr>
<th>Value</th>
<th>Name</th>
<th>Publicly Verifiable</th>
<th>Public Metadata</th>
<th>Private Metadata</th>
<th>Nk</th>
<th>Nid</th>
<th>Reference</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0001</td>
<td>VOPRF (P-384, SHA-384)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>48</td>
<td>32</td>
<td>Section 5</td>
<td>None</td>
</tr>
<tr>
<td>0x0002</td>
<td>Blind RSA (2048-bit)</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>256</td>
<td>32</td>
<td>Section 6</td>
<td>The RSABSSA-SHA384-PSS-Deterministic and RSABSSA-SHA384-PSSZERO-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Deterministic variants are supported</td>
<td></td>
</tr>
</tbody>
</table>

### 8.3. Media Types

This specification defines the following protocol messages, along with their corresponding media types:

*TokenRequest: "application/private-token-request"

*TokenResponse: "application/private-token-response"

The definition for each media type is in the following subsections.

#### 8.3.1. "application/private-token-request" media type

**Type name:** application

**Subtype name:** private-token-request

**Required parameters:** N/A

**Optional parameters:** None

**Encoding considerations:** only "8bit" or "binary" is permitted

**Security considerations:** see Section 7

**Interoperability considerations:** N/A

**Published specification:** this specification

**Applications that use this media type:** N/A

**Fragment identifier considerations:** N/A
8.3.2. "application/private-token-response" media type

Type name: application

Subtype name: private-token-response

Required parameters: N/A

Optional parameters: None

Encoding considerations: only "8bit" or "binary" is permitted

Security considerations: see Section 7

Interoperability considerations: N/A

Published specification: this specification

Applications that use this media type: N/A

Fragment identifier considerations: N/A

Additional information:

Magic number(s): N/A

Deprecated alias names for this type: N/A

File extension(s): N/A

Macintosh file type code(s): N/A

Person and email address to contact for further information: see Authors' Addresses section

Intended usage: COMMON

Restrictions on usage: N/A

Author: see Authors' Addresses section

Change controller: IESG
9. References

9.1. Normative References


9.2. Informative References


[CONSISTENCY] "*** BROKEN REFERENCE ***".


Appendix A. Acknowledgements

The authors of this document would like to acknowledge the helpful feedback and discussions from Benjamin Schwartz, Joseph Salowey, Sofia Celi, and Tara Whalen.
Appendix B. Test Vectors

This section includes test vectors for the two basic issuance protocols specified in this document. Appendix B.1 contains test vectors for token issuance protocol 1 (0x0001), and Appendix B.2 contains test vectors for token issuance protocol 2 (0x0002).

B.1. Issuance Protocol 1 - VOPRF(P-384, SHA-384)

The test vector below lists the following values:

*skS: The encoded OPRF private key, serialized using SerializeScalar from Section 2.1 of [OPRF] and represented as a hexadecimal string.

*pkS: The encoded OPRF public key, serialized using SerializeElement from Section 2.1 of [OPRF] and represented as a hexadecimal string.

*challenge: A random challenge digest, represented as a hexadecimal string.

*nonce: The 32-byte client nonce generated according to Section 5.1, represented as a hexadecimal string.

*blind: The blind used when computing the OPRF blinded message, serialized using SerializeScalar from Section 2.1 of [OPRF] and represented as a hexadecimal string.

*token_request: The TokenRequest message constructed according to Section 5.1, represented as a hexadecimal string.

*token_request: The TokenResponse message constructed according to Section 5.2, represented as a hexadecimal string.

*token: The output Token from the protocol, represented as a hexadecimal string.
B.2. Issuance Protocol 2 - Blind RSA, 2048

The test vector below lists the following values:

*skS: The PEM-encoded PKCS#8 RSA private key used for signing tokens, represented as a hexadecimal string.

*pkS: The DER-encoded SubjectPublicKeyInfo object carrying the public key corresponding to skS, as described in Section 6.5, represented as a hexadecimal string.

*challenge: A random challenge digest, represented as a hexadecimal string.

*nonce: The 32-byte client nonce generated according to Section 6.1, represented as a hexadecimal string.

*blind: The blind used when computing the blind RSA blinded message, represented as a hexadecimal string.

*salt: The randomly generated 48-byte salt used when encoding the blinded token request message, represented as a hexadecimal string.

*token_request: The TokenRequest message constructed according to Section 6.1, represented as a hexadecimal string.
*token_request: The TokenResponse message constructed according to Section 6.2, represented as a hexadecimal string.

*token: The output Token from the protocol, represented as a hexadecimal string.
challenge: 3f5a1c30d13f860622458ce836d8af325378054370fe8a3d771ebeb67d4d810d
nonce: c0fcbbb243d8f5d4f661dbdefca95879b39aeccb77b7db731b59c09688773125
blind: 0d00c7c076d18b4b201b4b4defc09d42bc09d3c4926c656b85956e08927590d2e4b34
570105ae492f655d41a3d6f8f1cc6a9a2895c3f6d45c88239257f2e6cee5bd887e7d87f635
67069d78f8bb95947c7ab123b166c9f3b76d856112802dd0fefa80a9c3887fbb5d949481b
4f7a21a0269f1761193dafa7197e8efc9efc9c2df0f861991977cdf01284083435f2f3df
f8ae2935ae0f5440b3b4ac12fed83a03bc499aba87241d62d2db0c6a64422eb63dbf
ba0193161648e582bdf319027d80a0e2953326fff0a9641500122ba81cf59077e
ebd4d2384221ddb99439c2465138b98348b58af89b4e65b7056a27e1f5308512e368c
fe6df4efc37597ed
salt: 4daf07bc96a829736ce6368a4d3ed989192ea4f0acb3ed715dca2ae868c16ad346
ee5e2b3d26eb4f68369a773e83bc5d
token_request: 0002ca832ffabbdd44e2cd54e5e24d74519d2976808ac9aba88e26b732
adcb382781e7e2657c9b9475bf9a6b2d02b3ec38b80d4e9627d5b62a7f1bea16b81e
46f3f5637ca4af93df5959f8a7dcac1ba5f8b685dab32a676213686c1121974df84e5
241c359299cb5fc41182b6d2b112f35a4073d1231299447f888488ba84eb5b4602534
787aa1e167bc1dccbce7bf5a4d3e2b24f2d4b939493987cfb911cf6f3785847deaaca6
350c16cb0b7882ec07ade7c781a3ade7c76366ee67ec23c9aaf8e4a8416b29e4c6aa888f2873
ba36ce6e2b9596aa508be34543a469286be2404f1f81f6a274a2af4292d6237777ab6d
e56379d24c72f705e3c7f1c3d150d
token_response: 6e7d5334765e4b44e4a4e3b81ae8f4f1334dfdc47b3dfaae2b3c899f42a
67d8239592ac4fa129a938e139bf952d8504bdadb90f7f54fda3d6e6efea6a0c15a0a50
fb2987b53d0558ed32d68b3536f6c9b953dabcf0f2ef6baf33c61286f607f0786190
6a2f6d919693140751a8e2173e67456f7f4ebe7ad0ee5c65ce82ad477d34e4b3755bdc
0f168ab85ce6626f87c56349be9363826dad4ab870ab975e8b8fdd0b95bcf457dc83377ff
ea8b5bc77d4d45cb4bdc5aeefcf958cc822cc53ded3da699af86bfad3054fe49da8ebeeb5
162e444e3b4d438f9e63ebadd50cba56bf43f0718a65e7d8dfc40762cdb9962edc731f6a7
e8c641bf9890ac9cd8f7b7f0

token: 0002c0fcbbb243d8f5d4f661dbdefca95879b39aeccb77b7db731b59c09688773125

125ad76b53adcc4a4e48a91eb99f3f9c933212f4aaeef07cb709e64da686a7ca572
f89829ca2a48a3056186322d9ca147266121ddeb5632c07f17f1cd2780dedf816364ed5
4d516d2fe803666e56ec1d4e8a0da7aed2675c15156d774b31178901bf52aae9926156
2289459a41c5739dec6d4c2447744fe07c53c9d090f053263d019255c6cfc27739132bd68
21ad49f1aa8db6873319d4c04760374a8f1d0806b2a25b4626465cb2ff972463b03152
589068389df894946d823b92ce73a9f6ebc1fed9cbf26d6fbae1ff3ef9f2d0267cd2
d5fcbca30f8b0ce09d9a1a39a40156b061403d05999a36f805347681ae5f02f3d081b36
cd797fe1a4df1ca9694320fc44c6cc7c5d90aaedc915af3ac11a3afaf562d38c8213e39f6
731fa5e701697d0bf7bfcfc83b447945b35115a20770370226b52a19df939f3080e
Authors' Addresses

Sofía Celi
Brave Software
Lisbon
Portugal

Email: cherenkov@riseup.net

Alex Davidson
Brave Software
Lisbon
Portugal

Email: alex.davidson92@gmail.com

Armando Faz-Hernandez
Cloudflare
101 Townsend St
San Francisco,
United States of America

Email: armfazh@cloudflare.com

Steven Valdez
Google LLC

Email: svaldez@chromium.org

Christopher A. Wood
Cloudflare
101 Townsend St
San Francisco,
United States of America

Email: caw@heapingbits.net